27428/36596

For Filing Sec.371 National Stage Only
"Application With Literal Translation"
534 Rec'd PCT/PTO 28 JUL 2000

PCT/DE99/00213

5/PRTS

COMPRESSION OF DATA WITH SYNTACTIC STRUCTURE

1. Introduction to the General Topic

The coding of source files has important technical applications in the area of transmission and storage of information. The desire to have a low data rate in the transmission of information, and a small space requirement in the storage of information, has given rise to the demand for a form of information coding that is as brief as possible, i.e. one with little or, ideally, no redundancy.

2. State of the Art

2.1. Forms of Redundancy

Redundancy in streams of source symbols takes two basic forms: it can appear as a differential frequency of occurrence of the elements in the set of source symbols, and in the form of statistical dependencies between temporally separated source symbols, which usually are caused by source-specific *formation laws* for the construction of a text as a sequence of source symbols.

3. Disadvantages of Known Methods

For the compression of messages from sources that suffer from redundancy and emit source symbols that are not uniformly distributed but are statistically independent, a procedure according to Huffman [1] is known. Being specially intended for this type of source, the

procedure enables maximal compression only under the extremely favorable condition that the statistical properties of the source are known or can be reliably estimated.

Whereas estimation of the probabilities of occurrence of individual source symbols, which is necessary for Huffman coding, can be done with sufficient accuracy even in the case of moderately long texts, the estimation of statistical dependencies between temporally separate source symbols as a rule requires extensive observation of the source, which goes far beyond what is feasible both technically (storage capacity) and in terms of observation time. For this reason the known or similar procedures always involve making assumptions about the underlying formation law or crucial characteristics thereof, and then of course are suitable only for sources with this formation law. This qualification also applies to the Lempel-Ziv (LZ) procedure for data compression [2], which has been known since the end of the 1970's. This procedure is suitable for the compression of arbitrary linear symbol sequences; that is, it can be applied with no knowledge of the statistical properties of the information source. The main areas of application today are the compression of pictorial data (GIF file format) and arbitrary files (archiving programs gnuzip, pkzip). In contrast to Huffman coding, this procedure utilizes statistical dependencies between temporally successive symbols, by taking into account repeatedly occurring subsequences of symbols (strings). Hence LZ compression is suitable only for sources, the formation law of which leads to the frequent occurrence of particular strings. The

redundancy introduced into a text by following grammatical rules during its construction is not accessible to the known compression procedures because of the recursive structure of grammatical rules, since these procedures merely employ symbol frequencies or repetitive chains of alphanumeric characters for compression.

4. Definition of the Object of the Invention

The object of the present invention is to provide a method and apparatus for implementing the method that enable a text consisting of a stream of alphanumeric characters, and is constructed according to grammatical rules, to be encoded in such a way as to remove the redundancy produced as a result of the limitation of all possible character sequences by the grammar.

This object is achieved by a method with the characteristics given in Claim 1 and two kinds of apparatus by means of which the characteristics given in Claims 8 and 9 are achieved.

5. Advantages of the Solution in Accordance with the Invention

With the solution in accordance with the invention, enabling data reduction in dependence on the programming-language syntax employed, it now becomes possible in an especially advantageous manner to remove, asymptotically almost completely, redundancy produced in texts during their construction because of the need to follow certain grammatical rules.

The invention has proved this by analytical calculation of the capacity of its data stream, encoded in a syntax-directed manner, using the example of a language with characteristics typical of programming languages (mathematical expressions, if-then-else construct, etc.). In this process it was shown that an appreciable amount of the redundancy that conventional methods (Lempel-Ziv, Huffman) are designed to address has not previously been eliminated by these methods.

The new method of syntax-directed coding, in contrast, is capable of reducing the amount of data by an additional *ca.* one-half in comparison, for instance, to a result obtained with the Lempel-Ziv compression procedure.

Thus the method in accordance with the invention, together with the apparatus specified for its implementation, is predestined for all applications that involve the transmission or storage of syntactically structured texts. Examples include the transmission of Java applets in the Internet or an intranet, the transmission or storage of Postscript files, the transmission and storage of MPEG4-encoded video data, and the use of higher protocols in communication. Especially for the transmission of programs, such as Java applets, syntax-directed coding (SDC) is particularly suitable because the parser is a component of both a compiler and an SDC coder, so that SDC can be organically integrated into the data flow from the writing to the running of the program.

6. Mode of operation of the invention

6.1 Syntactically structured source

The invention takes as a point of departure a formal language represented by a set of chains of alphanumeric characters. The text is put together according to a grammar assigned to the language. A grammar is both the mathematical system used to define a formal language, and a set of rules to determine the syntactic validity of a specific sentence. Only context-free grammars are considered here, because higher programming languages (C, C++, Java, etc.) are described almost exclusively by context-free grammars. The following definitions apply:

Definition 1 *A context-free grammar is a 4-tuple* $G = (N, T, P, S)$, *where*

N *is the set of non-terminal symbols,*

T *is the set of terminal symbols, and*

P *is the relation* $N \times (N \cup T)^*$.

S *is a special non-terminal symbol, also called start symbol.*

Definition 2 *An element* $(A, \beta) \in P$ *is called* production *and is abbreviated* $A \rightarrow \beta$.

A production (or derivation) is thus a replacement rule for a non-terminal symbol A, such that this non-terminal symbol is replaced by a string (chain) β composed of (non-terminal and) terminal symbols.

When there exist several productions (A, β_1), (A, β_2),

..., $(A, \beta_n) \in P$, for this we write:

$$A \rightarrow \beta_1 \mid \beta_2 \mid \beta_n$$

Definition 3 *The set* $P_{A0} \subseteq P$ *of all possible derivations for a non-terminal symbol* A_0

$$P_{A0} = \{(A,\beta) \in P : A = A_0\}$$

is also called the set of the alternatives *for* A_0. *The decision to use a specific production* $(A_0,\beta_k) \in P_{A0}$, $(k = 1,2, \ldots, n)$ *is called* selection.

Example 1 *All arithmetic expressions in the variables* a *with the operations* + *and* * *as well as arbitrarily nested brackets* (,) *are a formal language generated by the grammar*

$$G = (\{E,T,F\},\{a,+,*,(,)\},P,E).$$

P consists of the following productions:

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow (E) \mid a$$

a * a + (a * a + a) * a, *for example, is a valid sentence.*

In generating a sentence, further strings of non-terminal and, where appropriate, terminal symbols are derived by application of suitable productions to the

non-terminal symbols until ultimately the string consists only of terminal symbols. In the example given above, the valid sentence consists only of the terminal symbols a, +, *, (,).

6.2 Principle of Syntax-Directed Coding

Information always flows into a text being generated when during its construction *decisions* are made. Decisions are to be made when a particular one of several possible source symbols is to be chosen or, in the case of a syntactically structured source, out of several applicable productions $A \rightarrow \beta_1$, $A \rightarrow \beta_2$, ... precisely one production actually is applied (selection, cf. Definition 3).

SDC encodes, instead of the string of terminal symbols, the selections in the order in which they were made during the construction of this string (the text).

Definition 4 $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ *is called SDC alphabet, with*

$$n = \max_A \{|P_A|\}$$

All the elements in each set of alternative P_a are numbered (arbitrarily) with SDC symbols $\sigma_i \in \Sigma$:

Definition 5 $z_A : P_A \rightarrow \Sigma$ *is an (arbitrary) injective mapping.*

The concrete definition of z_A can in general be dependent

on P_A.

As an illustration of the invention the following exemplary embodiments are described in detail and shown in Figures 1 to 6, as follows:

Figure 1 is a block diagram of an SDC transmission system

Figure 2 shows alternatives of the grammar *G* and selections to which SDC symbols have been attributed

Figure 3 is a parse tree of the sentence a * a + (a * a + a) * a with a resulting linear SDC symbol sequence 1212221211212222222

Figure 4 is a flow diagram of the stack machine

Figure 5 is an example of an SDC transmission system with receiver-side source-text output

Figure 6 is an example of an SDC transmission system with executable Java applet or application as output

The functioning of a transmission system with SDC is illustrated in Figure 1. In principle transmission and storage are equivalent with respect to the way the coding works.

The source delivers a syntactically structured stream of symbols (source program). An elementary component of the SDC encoder is a parser, the task of which is to

retrieve from the initially linear sequence of source symbols the grammatical structure of the sentence (the source program). This grammatical structure is represented in general by a parse tree, which in turn serves as input to the encoder. The encoder encodes the parse tree to generate a linear sequence of SDC symbols, such that at each moment the current SDC symbol is output and transmitted in dependence on the current selection at that moment. The SDC decoder first reconstructs the parse tree from the linear sequence of SDC symbols and then, by traversing the parse tree, reproduces the sequence of terminal symbols sent out from the source.

6.3 Encoding

The elementary task of the parser comprises the reconstruction of the *selections* made by the source during construction of the sentence. A parsing step thus amounts to determining the specific production $(A, \beta) \in P_A$ that the source had selected for constructing the text at this point. As stated in Definition 5, the encoder assigns to the selection thus determined the associated SDC symbol $\sigma_k = z_A((A, \beta))$ and attributes σ_k to the corresponding node in the parse tree.

By traversing the parse tree, ultimately the linear sequence of SDC symbols is generated and output. In the examples given here, a depth-first algorithm is always used for this purpose. The process of encoding will now be explained in greater detail with reference to an exemplary parse tree and the underlying grammar from

Example 1.

Example 2 *The context-free grammar of Example 1 is assumed. All alternatives, i.e. all possibilities for deriving each non-terminal symbol, are shown graphically in Figure 2. Because the greatest set of alternatives comprises precisely two elements, the SDC alphabet* $\Sigma = \{1, 2\}$ *suffices. To each selection of all quantities of alternatives an SDC symbol is attributed. For example, the selection of the specific production* $T \rightarrow T * F$ *is assigned the SDC symbol 1.*

A valid sentence in this grammar is thus, for example: a * a + (a * a + a) * a. *The parse tree for this sentence and the generation of the SDC symbol sequence by depth-first traversing are shown in Figure 3.*

6.4 SDC with arithmetic encoding

The method that has just been described is efficient only if

1. $|P_A| = const \ \forall \ A$, and
2. $P((A_1 \ \beta_i)|A) = const \ \forall \ A, \ \beta_i$,

where $P((A_1 \ \beta_i)|\alpha)$ is the conditional probability of making the selection that generates β_i from the set of alternatives P_A.

This deficiency can be alleviated by arithmetic encoding [3] of the SDC alphabet Σ as stated in Claims 6 and 7. The probability distributions of the SDC symbols used

for arithmetic encoding should be adapted in an adapter according to Claim 10 in which case the adaptation must be carried out separately for each set of alternatives. Instead of the linear sequence of SDC symbols, the bit stream generated by the arithmetic encoder is transmitted.

6.5 Decoding

Decoding is the inverse of the encoding process. The decoder formats the SDC code back into the original sentence (strings of terminal symbols). It operates according to the procedure described in Claims 1, 3 and 4, 5 and is implemented by a stack machine [4] according to Claims 8 and 9.

A flow diagram for a method according to Claim 22 is given in Figure 4.

To explain operation of the stack machine the following abbreviations are employed:

β_k String of terminal and non-terminal symbols

σ SDC symbol

V Terminal or non-terminal symbol

S Start symbol (non-terminal symbol)

The decoding thus proceeds in the following steps (the numbers correspond to the reference numerals shown in Figure 4):

1. The development of a valid sentence according to the grammar (a program) begins with the start

symbol S. The start symbol S is placed on the stack.

2. The uppermost symbol V is read from the stack.
3. Check whether V is a non-terminal symbol.
4. From the linear input stream of SDC symbols the next SDC symbol σ is read.
5. In case V is a non-terminal symbol, the derivation with V is further developed. By way of σ a selection $(V, \beta_k) = z^{-1}(\sigma)$ is made for V from the set of alternatives of $V \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$. In accordance with the specific selection, V is then replaced by the string β_k.
6. The string β_k of symbols (terminal and/or non-terminal symbols) is placed on the stack.
7. If V is a terminal symbol, it is output.
8. Check whether the stack is empty.
9. The stack machine terminates when the stack is empty.

The strings of terminal symbols output successively, when linked together, finally yield the sentence originally encoded from the source.

7. Exemplary Embodiment: Preferred Form of Application

The two exemplary embodiments shown in Figures 5 and 6 differ merely in the form in which data are sent to the receiver (output of source text corresponding to the method according to Claim 3 versus direct output of an executable Java applet or an executable Java application, corresponding to the method according to

Claim 4).

1. Java source
   A Java source corresponds to an ASCII text based on the Java grammar, which consists of Java program text plus comments, blank spaces etc.

2. SDC encoder
   The SDC encoder is composed of a compiler front end ((3),(4) and(6)),a parse-tree processor (5), an adapter (8) and modules for compression and concatenation of the generated data.

3. Scanner
   The scanner reads the source text, removes the text sites that are unimportant for the program such as documentation, comments, blank spaces etc. and sends (terminal) symbols to the parser (4).

4. Parser
   The parser interprets the (terminal) symbols from the scanner (3), tests whether the symbol sequences agree with the grammatical rules for Java and sets up the specific parse tree for this program.

5. Parse-tree processing
   During the initialization of parse-tree processing all nodes of the parse tree are assigned SDC symbols specified for each type of node. The subsequent traversing of all nodes of the parse tree causes an SDC symbol to be output at each crossing of a node. These SDC symbols are sent to both the adapter (8) and the arithmetic coder (10). In addition, the parse-tree processor transmits the

current node type to the adapter (8).

6. Symbol-table memory
The symbol-table memory contains the identifiers of the Java program text extracted by the parser (4).

7. Zip-compression device
The content of the symbol table (6) is compressed by a method according to Lempel-Ziv-Welch [2].

8. Adapter (transmitter-side)
During the process of initialization the adapter enters initial probability distributions for each node type into the probability-distribution table (9). During processing of the SDC symbols received from the parse-tree processor (5), and with reference to the current node type, at each step the probability distribution of all SDC symbols of the current node type is adapted and entered into the probability-distribution table (9). The adapter 8 simultaneously makes available to the arithmetic encoder (10) the probability distribution associated with the SDC symbol currently being processed.

9. Table of the probability distributions of the SDC symbols
For each node type, the table contains a probability distribution of SDC symbols that is peculiar to that node type.

10. Arithmetic encoder
The arithmetic encoder (3) receives from the parse-

tree processor (5) the next SDC symbol to be encoded and encodes it with reference to the probability distribution made available by the adapter (8). The output of the arithmetic encoder is a binary data stream.

11. Data concatenation
The binary data stream from the arithmetic encoder (10) and the symbol table (6) compressed by the compressor (7) are combined to form a data packet.

12. HTTP server
On the HTTP server the data packet created in the data concatenation (11) is deposited and made available to be called up from a mass storage device.

13. Intranet/Internet

14. HTTP client (Internet browser)
The browser has the task of downloading the data packet from an HTTP server (12) by way of the intranet/Internet (13) and initiating the process of decoding by the SDC decoder (15).

15. SDC decoder
The SDC decoder comprises substantially of the arithmetic decoder (17) and the stack machine (22). The result of its operation is the sequence of (terminal) symbols that correspond to the original Java source code (1) (cf. 3 and 4).

16. Data extractor

Here the binary data stream from the arithmetic encoder (10) and the symbol table compressed by the zip compressor (7) are extracted from the common data packet and sent on, the former to the arithmetic decoder (17) and the latter to the zip-decompressor (18).

17. Arithmetic decoder
The arithmetic decoder [3] adaptively decodes the binary data stream produced by data extraction (16), taking into account the probability distributions associated with the SDC symbols that have been made available by the receiver-side adapter (20), and at each decoding step transmits an SDC symbol to the stack machine (22).

18. Zip-decompressor
The symbol table from the coder (6), compressed by the Lempel-Ziv method (7) and received from the data extractor (16), is unpacked and entered into the symbol-table memory of the decoder (19).

19. Symbol-table memory of the decoder

20. Adapter (receiver-side)
The initialization of the receiver-side adapter 20 is analogous to that of the transmitter-side adapter (8). While decoding proceeds, at each decoding step the adapter receives an SDC symbol, adapts the probability distribution of the SDC symbol of the current node type and enters the adapted probability distributions into the probability-distribution table (21). At the same

time, the receiver-side adapter supplies to the arithmetic decoder the probability distribution that is valid for the next decoding step. The selection of this distribution is determined by the node type determined by the stack machine (22) and expected in the next decoding step.

21. Table of probability distributions of the SDC symbols
For each node type the table contains an individual probability distribution of the SDC symbols of this node type.

22. Stack machine
The initialization of the stack machine, the processing of the stack memory and the output of terminal symbols is represented in Figure 4 as a flow diagram. In addition the stack machine transmits the following data to the adapter(20):

(a) the current node type, for adaptation of the probability distributions of the SDC symbols of this node type

(b) the next node type, the SDC-symbol probability distribution of which the adapter (20) must supply to the arithmetic decoder (17) in the next decoding step.

23. Stack memory
Stack memory for the stack machine.

24. Java sink

Retrieved Java program text without comments, superfluous blank spaces etc.

25. Code generator

The code generator corresponds to a compiler back end and generates byte code for the JVM (26) in dependence on the sequence of (terminal) symbols provided by the stack machine (22).

26. JVM

The virtual machine for Java programs (Java Virtual Machine).

References

[1] Huffman, D.A.: *a Method for the Construction of Minimum-Redundancy Codes*. Proc. IRE, 40:1098-1101, 1952.

[2] Ziv, Jacob and Abraham Lempel: *A Universal Algorithm for Sequential Data Compression*. IEEE Trans. Inform. Theory, IT-23(3):337-343, May 1977.

[3] Witten, Ian H., Radford M. Neal and John G. Cleary: *Arithmetic Coding for Data Compression*. Commun. ACM, 30(6):520-540, June 1987.

[4] Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman: *Compilerbau*. Addison-Wesley Publishing Company, Bonn, 4th Edition, 1988.

Key to reference characters

Reference characters in Figure 4

β_k String of terminal and non-terminal symbols

σ SDC symbol

V Terminal or non-terminal symbol

S Start symbol (non-terminal symbol)

1. The development of a valid sentence according to the grammar (a program) begins with the start symbol S. The start symbol S is placed on the stack.
2. The uppermost symbol V is read from the stack.
3. Check whether V is a non-terminal symbol.
4. From the linear input stream of SDC symbols the next SDC symbol σ is read.
5. If V is a non-terminal symbol, the derivation is continued with V. By means of σ a selection $(V, \beta_k) = z^{-1}(\sigma)$ from the set of alternatives of $V \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$ is determined for V. According to the particular selection made, V is replaced by the string β_k.
6. The string β_k of symbols (terminal and/or non-terminal symbols) is placed on the stack.
7. If V is a terminal symbol, it is output.
8. Test whether the stack is empty.
9. The stack machine terminates when the stack is empty.

Reference characters in Figures 5 and 6

1. Java source
2. SDC encoder
3. Scanner
4. Parser

5. Parse-tree processor
6. Symbol-table memory
7. Zip-compression device
8. Transmitter-side adapter
9. Table of probability distributions of the SDC symbols
10. Arithmetic encoder
11. Data concatenation
12. HTTP server
13. Intranet/Internet
14. HTTP client (Internet browser)
15. SDC decoder
16. Data extractor
17. Arithmetic decoder
18. Zip-decompressor
19. Symbol-table memory of the decoder
20. Adapter (receiver side)
21. Table of probability distributions of the SDC symbols
22. Stack machine
23. Stack memory
24. Java sink
25. Code generator
26. JVM